Using Semantic Wikis to Support Software Reuse

Sajjan G. Shiva
The University of Memphis/Department of Computer Science, Memphis, TN, USA
Email: sshiva@memphis.edu

Lubna A. Shala
The University of Memphis/Department of Computer Science, Memphis, TN, USA
Email: lshala@memphis.edu

Abstract— **It has been almost four decades since the idea of software reuse was proposed. Many success stories have been told, yet it is believed that software reuse is still in the development phase and has not reached its full potential. How far are we with software reuse research? What have we learned from previous software reuse efforts? This paper is an attempt to answer these questions and propose a software reuse repository system based on semantic Wikis. In addition to supporting general collaboration among users offered by regular wikis, semantic Wikis provide means of adding metadata about the concepts and relations that are contained within the Wiki. This underlying model of domain knowledge enhances the software repository navigation and search performance and result in a system that is easy to use for non-expert users while being powerful in the way in which new artifacts can be created and located.**

Index Terms— **Software Reuse, Architecture, Domain Engineering, Product Lines, Component, Software Repository, Wiki, Knowledgebase, Metadata, Semantic, Ontology, Semantic Search, Retrieval System.**

I. INTRODUCTION

The gap between the rising demands of complex software systems and the ability to deliver quality software in a timely and cost effective manner keeps increasing. This has resulted in a great pressure to improve productivity and efficiency of software development. Several years of research in software engineering have suggested that the key to speed up software development and reduce its cost is to avoid "reinventing" existing software artifacts by reusing previously developed software when possible.

Software Reuse is defined as the process of building or assembling software applications and systems from previously developed software. It has been associated with software engineering since its early days as the NATO Software Engineering Conference in 1968 marked the birth of the idea of systematic software reuse as well as the formal birth of the field of software engineering. Since then, reuse has been a popular topic of research and debate for almost forty years in the software community.

The notion of software reusability is not a new idea; it has been practiced informally since the early days of software development. Many developers have successfully reused known algorithms and sections of code from older programs. However, this is usually done informally when opportunities arise, i.e., *opportunistic* reuse. While opportunistic reuse might work well for individual or small groups of programmers, it is very limited and dependent on these programmers' skills. The full benefit of software reuse can only be achieved by *systematic* software reuse that is conducted formally as an integral part of the software development cycle by constructing and applying multi-use artifacts such as architectures, patterns, components, and frameworks. Although a potentially powerful technology, systematic software reuse is still uncommon in the corporate world. Even though several major corporations have adopted systematic reuse programs and despite the advances in reuse enabling technology, systematic reuse in practice is still a difficult goal to accomplish. Therefore, the search for tools and technologies to promote successful and productive reuse is still an active area of research.

Numerous approaches to successful systematic software reuse have been proposed over the past four decades. Four of these main approaches are described in the next few sections of this paper. Several major corporations such as Hewlett-Packard and AT&T have successfully adopted software reuse. A few of these reuse successful stories in industry are discussed in a later section.

The final section of the paper presents our proposal to a software reuse repository system that is based on semantic wikis. A semantic wiki is a structured wiki that is enhanced with semantic web technologies by the addition of an underlying model of its page content knowledge (ontology). It combines the strength of easy to use and collaborative wiki pages, and machine accessible semantic web. A software reuse system that is based on semantic wikis provide a scheme where software developers and engineers can collaborate together to build a software repository that is easy to maintain and search which can lead to an effective systematic reuse.

Based on "Software Reuse: Research and Practice", by Sajjan Shiva and Lubna Shala which appeared in the Proceedings of the Fourth International Conference on Information Technology: New Generations, ITNG '07, April 2-4 2007, Las Vegas, Nevada, USA. © 2007 IEEE.

© 2008 ACADEMY PUBLISHER

II. Different Approaches to Software Reuse

Since the concept of systematic software reuse was proposed in 1968, several approaches have been suggested to achieve the promised potential of software reuse. Four of the major approaches are component-based software reuse, metadata-based software reuse, software architecture and design reuse, and domain engineering and software product lines. Component-based software development approach is based on the idea that there are so many similar components in different software systems that new systems can be built more rapidly and economically by assembling components rather than implementing each system from scratch. Metadata-base reuse relies on the idea of augmenting components with additional information about the component itself (metadata). Architecture-based reuse extends the definition of reusable assets to a whole design or subsystem composing of components and relationship among them. Domain engineering captures the commonalities and variabilities in a set of software systems and uses them to build reusable assets. The three approaches are not mutually exclusive and in many cases a combination is used. The following sections briefly survey each of these approaches and some of their common methods and techniques.

III. Component-Based Software Reuse

The notion of building software from reused components the same way electronic circuits are built from prefabricated ICs was first published in the NATO conference in 1968. The idea emerged from object-oriented development failure to support systematic reuse, which needed more abstract components than detailed and specific object classes. Component-based development (CBD) allows the reuse of large and highly abstract enterprise components so fewer components are needed to build the application. This reduces the assembling and integration efforts required when constructing large applications [1].

A software component is a prewritten element of software with clear functionality and a well-defined interface that identifies its behaviour and interaction mechanism. McClure [1] identifies several properties a software component is expected to have to be reusable. These properties include a set of common functionality, a well-defined interface that hides implementation details, the ability to inter-operate with other components, and the ability to be reusable in several different software applications and systems

The biggest challenge facing component based software reuse is managing a large number of stored reusable components efficiently to allow fast allocating and retrieving. While all component-based development systems include a library or a repository to store pre-built components, the best way to structure these repositories and implement the search interface has not been agreed upon. Several methods have been proposed in literature since the early days of component reuse. Below is a summary of the most common approaches.

A. *Component Repository Structuring and Retrieval*

The repository structure is an essential factor in obtaining good retrieval results. Even though some retrieval algorithms can provide adequate effectiveness with minimal indexing and structuring efforts, poorly structured repository with insufficiently indexed components will not have a good retrieval performance regardless of the retrieval algorithm [2].

Software repository structuring (indexing) methods can be divided to two main categories: manual and automatic indexing. Common examples of manual indexing include enumerated and faceted classification. Automatic indexing most common method is free-text indexing.

Enumerated Classification involves defining a hierarchical structure of categories and subcategories where actual components are at the leaf level of the classification tree. The hierarchical structure is easy to use and understand and provides a natural searching method of navigating in the classification tree. However, the main problems with this classification include its inflexibility and difficulty to change as the indexing domain evolves. Additionally, it requires extensive domain analysis before classifying components into exclusive categories. Furthermore, retrieving classified components is only easy for users who understand the structure and contents of the repository; users unfamiliar with the structure will most likely be lost. Finally, Single classification can not represent complicated domains as no one classification is correct under all circumstances.

Faceted classification was proposed by Ruben Prieto-Diaz in 1987 [3]. It defines a set of mutually exclusive facets combine to completely describe all the components in a domain. Each facet can be described by a group of different terms. Users search for components by choosing a term to describe each of the facets. A faceted classification scheme gives freedom to create complex relationships by combining facets and terms. It is much easier to modify than a hierarchical classification because one facet can be changed without affecting others in the classification scheme. On the other hand, users must be familiar with the structure of the facets and their terms. Also, sometimes it is not obvious what combination of terms to use in the search.

Free-text indexing uses the words in a document to create index term lists that contain all words along with how many times they appear in the document. Most free-text indexing techniques work with a stop list that prevents certain high-frequency words such as “a”, “the”, and “is” from being indexed. To find a document that contains a set of keywords, users specify these keywords that are matched against the index term list to find the best matching documents. Even though easy to build and works very well in many applications including online search engines, free-text indexing is not suitable for indexing code and other software artifacts. Free-text indexing relies heavily on natural language rules and use statistical measures that need large bodies of text to be accurate. However code has arbitrary rules of grammar

© 2008 ACADEMY PUBLISHER

and allows the use of non-words and abbreviation and may contain minimum or no documentation [2].

B. *Repository Retrieval Issues*

Effective repository retrieval interfaces are essential to the success of any component-base development system. These interfaces, however, must overcome several issues before they can serve their purpose in supporting software reuse. One major issue is the reuse barrier. Most current reuse repository systems are designed as a separate process, independent of current development processes and tools assuming that software developers will start the reuse process when they need to. This assumption, however, is often incorrect. Several empirical studies suggested that "no attempt to reuse" is the most common failure mode in the reuse process. Because of the large and constantly changing component repositories, programmers often fail to correctly anticipate the existence of reusable components. When developers do not believe that a component exists, they will not even make an attempt to find it [4]. Therefore, the reuse process will never be initiated in the first place.

A second issue with repository retrieval systems is vocabularies and ill-defined queries submitted by non-expert users. This issue is a consequence of another faulty assumption repository retrieval systems make that developers know exactly what they need and can translate it into well-defined queries. Very often, searchers have a vague idea of what they need to find and can not express it in a clear set of terms. Even if users know what they are looking for, they may not have the knowledge needed to express it in the terms and vocabularies the retrieval system uses. The same person might use different terms to describe the same needed item at different times Furthermore, repositories are usually indexed by experts who might use terms that non-expert developers are not familiar with [2]. According to a study conducted in 2006, novice developers are more likely to reuse than expert ones [5]. Therefore, novice developers should be the target users for these systems.

C. *Existing Retrieval Tools*

Several tools have been proposed in literature to facilitate software reuse process. Expert system based reuse tools were developed in mid 1990s [6]. Many other tools were also proposed to address the retrieval issues described above. This section presents three of these tools: CodeFinder, CodeBroker, and Hipikat.

CodeFinder was first proposed by Henninger in 1995 [2] to support the process of retrieving software components when information needs are ill-defined and users are not familiar with terms used in the repository. It employs two techniques: intelligent retrieval method that finds information associatively related to the query, and query construction supported through incremental refinement of queries. The main idea is to create a retrieval method that utilizes minimal, low cost repository structure and evolves to adapt to user needs.

The proposed system is composed of three main parts: a tool (PEEL) to initially populate the repository with reusable components, a searching mechanism, and an adapting tool to refine the repository when needed.

PEEL (Parse and Extract Emacs Lisp) is a semi-automatic tool that translates Emacs Lisp (a variant of Lisp) source code files into individual reusable, components of functions and routines. These components are indexed in the repository using terms extracted from variable and function names and comments preceding definitions.

The system provides a user interface that implements the searching and browsing mechanism to allow the user to view and brows the hierarchy of the repository as well as submit search queries. Each time the user sends a query, the system responds with the retrieved items and a list of terms used to index each item. This technique helps users that are not familiar with the terms. As the user explores the information space, they become more familiar with repository structure and terms and incrementally refine their queries based on of previous results. To address the ill-defined query problem, the CodeFinder retrieval method does not try to exactly match the user queries with the item descriptions. Applying associative network, it uses association to retrieve items that are relevant to a query but don't exactly match it.

CodeFinder, however, does not address the reuse barrier issue. The user interface and tools are separated from the Lisp development environment. It relies on the developer to initiate the reuse process by switching between the two environments whenever needed. Many developers overestimate the cost of switching and searching for reusable item and do not attempt it. In 2001, Yunwen proposed an *active* and *adaptive* reuse repository system called "CodeBroker" to address this issue [4].

CodeBroker repository system is not a standalone tool; it is integrated into the development process running continuously in the background of the development environment. It is an active system that presents information when identifies a reuse opportunity without waiting for users to submit explicit queries. It is also an adaptive system that captures developer's preferences and knowledge level and saves them to a profile used to adapt the system's behaviour to each user. User profiles can be explicitly modified by users (adaptable) or implicitly updated by the system (adaptive).

The system's repository is composed of Java API and General Library. Its architecture consists of three software agents: listener, presenter, and fetcher. Listener is a background running process that captures developers' needs for reusable components and formulates reuse queries. Whenever a developer finishes a doc comment or a function definition, Listener automatically extracts the contents and creates a concept query and passes it to fetcher. Fetcher executes the query retrieving relevant components based on concept similarity to the comment text or constraint compatibility to the function signature. Presenter displays retrieved components to the user in the RCI-display taking into consideration user profile settings.

© 2008 ACADEMY PUBLISHER

A similar tool to CodeBroker called "Hipikat" was proposed by Čubranič et al. in 2005 [7]. Hipikat is essentially intended to assist software developers who join an existing software development team by recommending items from the project's development history including code, documentation, and bug reports that may be of relevance to their current task.

Hipikat tool performs two main functions: it builds a project memory from artifacts created as part of a software system development, and recommends items from the project memory that it considers relevant to the task being performed. These two functions are executed concurrently and independently. The construction of the project memory is a continuous process adding and updating items whenever new project information is created. Once the project memory updates are committed, they can be included in recommendations to users [7].

Like CodeBroker, Hipikat is designed to be integrated in the development environment platform. The tool is a part of an ongoing research project and its free prototype Eclipse plug-in can be downloaded online[1].

IV. Metadata-Based Software Reuse

Most software components including source code, test cases, and design models are not human readable which makes searching for these components in their repository harder than searching for text documents[8]. One way to make a software asset easier to find and retrieve from the repository is the use of metadata.

Metadata is simply data describing data. In case of software reuse, metadata is a kind of representation that describes a software asset from all aspects including how to use it and how it relates to other assets which helps locating an asset and determining if it is suitable to be used.

Reusable software component metadata are collected in metadata repositories which allows to proactively manage the metadata and make it available in a highly consumable manner. One example of a well-known metadata repository in industry is Logidex, a collaborative Software Development Asset (SDA) management tool that simplifies the creation, migration and integration of enterprise applications. SDA may contain executables such as software components and services, software development lifecycle artifacts, and knowledge assets such as best practices and design patterns. Logidex has often been used to store the organization reuse patterns and recommend candidate components for reuse.

V. Architecture-Based Software Reuse

Effective reuse depends not only on finding and reusing components, but also on the ways those components are combined [9]. This means reuse of the control structure of the application. System software architecture is composed of its software components, their external properties, and their relationships with one another. Architecture-based reuse extends the definition of reusable assets to include these properties and relationships.

Shaw classified software architecture into common architecture styles where every style has four major elements: components, connectors that mediate interactions among components, a control structure that governs execution and rules about other properties of the system, and a system model that captures the intuition about how the previous elements are integrated. Some of the popular architecture styles in [9] are pipeline, data abstraction, implicit invocation, repository, and layered.

Applying a combination of architecture styles create architectural patterns [10]. An architectural pattern is a high-level structure for software systems that contains a set of predefined sub-systems, defines the responsibilities of each sub-system, and details the relationships between sub-systems. Layers, Pipes and Filters, and Blackboard are some of the common patterns described by Buscmann et al in [11].

Software systems in the same domain have similar architectural patterns that can be describe with a generic architecture (domain architecture). Domain architecture is then refined to fit individual application in the domain creating application architecture [1][10].

VI. Domain Engineering and Software Product Lines

Domain engineering, use of specific knowledge and artifacts to support the development and evolution of systems, has become a major aspect of disciplined software reuse. Most organizations work only in a small number of domains. For each domain they build a family of systems that vary based on specific customer needs. Identifying the common features of existing systems within a particular domain and using these features as a common base to build a new system in the same domain may result in higher efficiency and productivity.

Domain engineering has two stages domain analysis and domain implementation. Domain analysis is the process of examining the related systems in a domain to identify the commonalities and variabilities. Domain implementation is the employment of that information to develop reusable assets based on the domain commonalities and use these assets to build new systems within that domain. Following is an overview of several known domain engineering approaches.

A. DARE

Domain Analysis and Reuse Environment (DARE) is a tool developed in 1998 to support capturing domain information form experts, documents, and code. Captured domain information is stored in a database (domain book) that typically contains a generic architecture for the domain and domain-specific reusable components. DARE also provides a library search facility with a windowed interface to retrieve the stored domain information [12].

[1] http://www.cs.ubc.ca/labs/spl/projects/hipikat/

© 2008 ACADEMY PUBLISHER

B. FAST

Family-Oriented Abstraction, Specification and Translation (FAST) is a system family generating method based on an application modeling language (AML). It was developed by Weiss and Lai at AT&T and continued to evolve at Lucent Technologies [13].

FAST is a systematic way of guiding software developers to create the tools needed to generate members of a software product line using a two-phase software engineering method: domain engineering phase and application engineering phase. The domain engineering phase defines the product line requirements and design through a Commonality and Variability Analysis and develops a small special purpose language to describe these commonalities and variabilities. The application engineering phase uses the application modeling language as the basis to generate the requirements and design of new family members.

C. FORM

Kyo C. Kang and others in Pohang University of Science and Technology presented a Feature Oriented Reuse Method (FORM) as an extension to the Feature Oriented Domain Analysis (FODA) method as FORM incorporates a marketing perspective and supports architecture design and object oriented component development [14]. FORM is a systematic method of capturing and analyzing commonalities and differences of applications in a domain (features) and using the results to develop domain architectures and components. It starts with feature modeling to discover, understand, and capture commonalities and variabilities of a product line.

D. KobrA

KobrA is a German acronym (**Ko**mponenten**b**asie**r**te **A**nwendungsentwicklung) stands for component-based application development [1]. The KobrA method offers a full life-cycle approach that integrates the advantages of several advanced software engineering technologies including product line development, component based software development, and frameworks to provide a systematic approach to developing high-quality, component-based application frameworks [15].

KobrA is based on the principle of strictly separating the product from the process. The products of a KobrA project are defined independently of, and prior to, the processes by which they are created, and effectively represent the goals of these processes. Furthermore, KobrA is "technology independent" in the sense that it can be used with all three major component implementation technologies CORBA, JavaBeans and COM.

E. PLUS

Product Line UML-Based Software Engineering (PLUS) is a model-driven evolutionary development approach for software product lines. It was introduced by Gomaa in 2004 as an extension to the single system UML-base modeling methods to address software product lines [16]. In addition to analyzing and modeling a single system, PULS provides a set of concepts and techniques to explicitly model the commonality and variability in a software product line. With these concepts and techniques, object oriented requirements, analysis, and design models of software product lines are developed using UML 2.0 notation.

VII. SOFTWARE REUSE IN INDUSTRY

Many organizations have been successful with software reuse. Hewlett-Packard (HP), for example, has a long history with different levels of software reuse started in 1989. It began with the development and networked distribution of a family instrument modules and evolved in 1991 to a corporate reuse program that guided several divisional reuse pilot projects for embedded instrument and printer firmware [17][18].

AT&T's BaseWorkX reuse program started in 1990 as an internal, large-grain component-reuse and software-bus technology to support telephone-billing systems. By 1995, AT&T was reusing 80 to 95% of its components [18].

A more recent example was implemented by ISWRIC (Israel SoftWare Reuse Industrial Consortium), a joint project of seven leading Israeli industrial companies [19]. It was a two-phase, three-year project started in 2000. During the first phase, a common software reuse methodology was developed to enable software developers to systematically evaluate and compare all possible alternative reuse scenarios. During the second phase, all seven participating companies implemented the methodology in real projects. Each company modified the model to better fit the specific needs of its pilot projects and evaluated the methodological aspects relevant to the pilot projects and the company.

Another industrial application for software reuse that is becoming more popular is global software distribution system development (GDSD). Skandia, one of the world's top life insurance companies, collaborated with Tata Consultancy Services (TCS) to create several IT-enabled financial services in different countries by integrating components developed independently by TCS and other third-party vendors at their own sites [20].

The appearance of some promising software reuse tools may lead to more successful industrial software reuse stories. One example is CodeSmith, the software generating tool that can produce code for any text-based language including C#, VB.NET, Java and FORTRAN. The tool has had a huge success in industry due to its advanced integrated development environment and its extensible, template-driven architecture that give developers full control over the generated code.

VIII.OUR RESEARCH

The focus of our software reuse research is to build an efficient component retrieval system that brings together the best of the systems previously discussed. Integrating the retrieval system with the development environment could greatly contribute to increasing developer's intention to reuse. Also employing software agents to create an "intelligent" system that learns to adapt to

© 2008 ACADEMY PUBLISHER

different users' knowledge level and terminology helps users find the needed components faster and makes the reuse experience more rewording to developers which encourage them to attempt it again. The repository systems should also incorporate machine-readable metadata which makes software components more manageable and improves the search performance and the overall value of the reuse library. However, metadata alone may not be sufficient to answer queries that require background knowledge about application domains or artifacts. Therefore, capturing that background knowledge using ontologies can significantly increase the system capability to support a wider range of reuse tasks [8]. Finally, we are investigating the possibility of further improving the search and retrieval process using Natural Language Processing (NLP) techniques.

In the following section we propose a framework for software reuse that incorporates the features presented above through the use of semantic wikis that enable efficient structuring and constant storage of the software artifacts, and provide a knowledge repository where content can be accessed and edited collectively using a web browser.

IX. SOFTWARE REUSE AND SEMANTIC WIKIS

Before describing the proposed system components, we introduce some of the important concepts and terminologies that will be used later in the discussion.

A. *Wikis*

In general, a Wiki is a web application designed to support collaborative authoring by allowing multiple authors to add, remove, and edit content. The word "Wiki" is a shorter form of "Wiki Wiki Web" derived from the Hawaiian expression "Wiki Wiki" which means "quick." [21]. Wiki systems have been very successful in enabling non-technical users to create Web content allowing them to freely share information and evolve the content without rigid workflows, access restrictions, or predefined structures.

Throughout the last decade, Wikis have been adopted as collaborative software for a wide verity of uses including software development, bug tracking systems, collaborative writing, project communication and encyclopedia systems[2]. Regardless of their purpose, Wikis usually share the following characteristics [21] [22]:

- **Easy Editing:** Traditionally, Wikis are edited using a simple browser interface which makes editing simple and allows to modify pages from anywhere with only minimal technical requirements.
- **Version Control:** Every time the content of Wikis is updated, the previous versions are kept which allows rolling back to earlier version when needed.
- **Searching:** Most Wikis support at least title search and some times a full-text search over the content of all pages.
- **Access:** Most Wikis allow unrestricted access to their contents while others apply access restrictions by assigning different levels of permissions to visitors to view, edit, create or delete pages.
- **Easy Linking:** Pages within a Wiki can be linked by their title using hyperlinks.
- **Description on Demand:** Links can be defined to pages that are not created yet, but might be filled with content in the future.

From the software reuse point of view, Wikis can be seen as a "lightweight platform for exchanging reusable artifacts between and within software projects" that has a low technical usage barrier[22]. However, using Wiki systems as a knowledge repository for software reuse has a major drawback. The growing knowledge in Wikis is not accessible for machines; only humans are able to read and understand the knowledge in the Wiki pages while machines can only see a large number of pages that link to each other. This problem may negatively affect the Wiki's searching and navigation performance.

B. *Semantic Wikis*[3]

A semantic Wiki is a Wiki that has an underlying model of the knowledge described in its pages. While regular Wikis have only pages and hyperlinks, semantic Wikis allow identifying additional information about the pages (metadata) and their relations and making that information available in a formal language (annotations) such as Resource Description Framework (RDF) and Web Ontology Language (OWL) accessible to machines beyond mere navigation. Adding semantics (structure) to Wikis enhances their performance by adding the following features [21].

- **Contextual Presentation:** Examples include displaying semantically related pages separately, displaying information derived from the underlying model of knowledge, and rendering the contents of a page in a different manner based on the context.
- **Improved Navigation:** The semantic framework allows relating concepts to each other. These relations enhance navigation by giving easy access to relevant related information.
- **Semantic Search:** Semantic Wikis support context-sensitive search on the underlying knowledge base which allows more advanced queries.
- **Reasoning Support:** Reasoning means deriving additional implied knowledge from the available facts using existing or user-defined rules in the underlying knowledgebase.

With these enhanced features, semantic Wikis can be valuable for software reuse. In addition to supporting general collaboration among users, semantic Wikis provide means of adding metadata about the concepts (artifacts) and relations that are contained within the Wiki. This system has the advantage of being easy to use

[2] http://www.wikipedia.org/

[3] Semantic Wikis are referred to in some literature as ontology-based Wikis or "Wikitologies."

© 2008 ACADEMY PUBLISHER

for non-expert users while being powerful in the way in which new artifacts can be created and stored [23].

C. Proposed System Components

We propose a framework for software reuse repository system that can stand alone or be integrated in a software development environment (IDE) enabling users to access and update a semantic repository of reusable software artifacts. The system is based on a semantic Wiki, which allows efficient structuring and storage of the software artifacts as well as reasoning about the repository's completeness and consistency. Another major advantage of using semantic wikis for software repository is that it allows for standard keyword search as well as faceted-based browsing which enables users to select artifacts according to certain facets.

The proposed system consists of the following components:

Repository: contains the stored reusable software artifacts.

User Interface: a web editor that enables users to manipulate the semantic repository adding and updating software artifacts as well as metadata about these artifacts.

Tools and Templates: to help users add and update metadata about software components.

Semantic Search: supports context-based search for relevant items.

NLP Component: enhances the searching and retrieval capabilities.

A simplified diagram of these components is shown in Fig. 1 below.

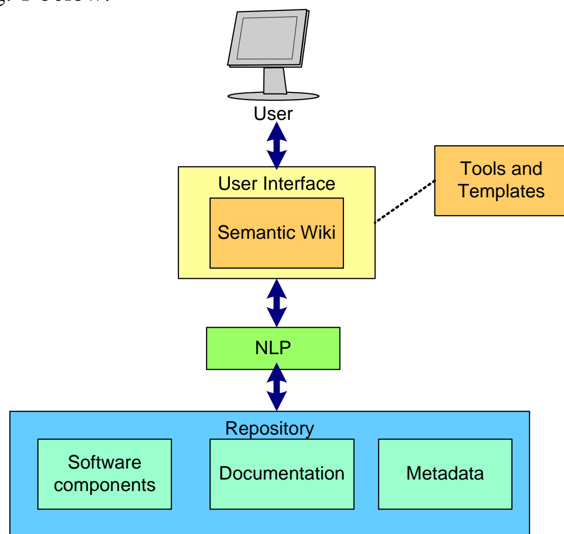

Figure 1. Proposed System.

X. Related Work

As a result of the increasing popularity of semantic wikis and their applications in the last few years, several software reuse systems have been proposed that are based on ontology or semantic Wiki.

The first one was presented in 2005 in [22] as *Self-organized Reuse of Software Engineering Knowledge Supported by Semantic Wikis* which was part of the RISE (Reuse in Software Engineering) project. It proposes an approach to support self-organized reuse by using semantic Wikis augmented with domain-specific ontologies to provide a means to manage the organic growth implied by the Wiki approach. Self-organized implies that the community does not only provide the artifacts to be reused, but also cares about how to organize them which distribute the effort for the "development of artifacts for reuse" within the community. The authors provide an application example for gathering and exchanging artifacts during the requirement phase of software development.

Another system was introduced in 2006 in [8] as KOntoR—an ontology-enabled approach to software reuse. The presented KOntoR architecture consists of two major elements: an XML-based metadata repository component to describe software artifacts independently from a particular format, and a knowledge component which comprises an ontology infrastructure and reasoning to leverage background knowledge about the artifacts.

Finally, one of the selected projects for Google's Summer Code program in 2007 is titled *Semantic-aware software component provisioning: actually reusing software*[4]. The project aims to "project aims to actually achieve software reuse in an effective, reliable and developer-friendly fashion." However, no architecture or implementation details are given.

XI. Conclusions

Software reuse is a longtime practiced method. Programmers have copied and pasted snippets of code since early days of programming. Even though it might speed up the development process, this "code snippet reuse" is very limited does not work for larger projects. The full benefit of software reuse can only be achieved by *systematic* reuse that is conducted formally as an integral part of the software development cycle.

This paper gives a summary of some important aspects of software reuse research and presents a rough proposal for a software reuse repository system that is based on semantic wikis. The next step will be to further research the concept and implement a prototype to ensure its validity.

Systematic software reuse is proposed to increase software development productivity and quality and lead to economic benefits. Several successful software reuse programs in industry have been reported. Yet, it is agreed upon that software reuse has not reached its full potential. Even though more and more organizations are adopting reuse successfully, systematic reuse has not yet become the norm. Reuse may not be the "silver bullet" that solves all software productivity problems, but a combination of technology and discipline can give reuse a chance to show its potential.

[4] http://wiki.eclipse.org/Semantic-aware_software_component_ provisioning:_ actually_reusing_software

© 2008 ACADEMY PUBLISHER

XII. References

[1] C. McClure. Software Reuse: A Standards-Based Guide, Wiley-IEEE Computer Society Pr, New York, 2001.

[2] Henninger, S., “An Evolutionary Approach to Constructing Effective Software Reuse Repositories,” ACM Trans. On Software Engineering and Methodology, 6(2), 1997, pp. 111-140.

[3] R. Prieto-Diaz. "*Classifying Software for Reusability*" IEEE Software, vol. 4, no. 1, 1987, pp 6-16.

[4] Ye, Yunwen. “An Active and Adaptive Reuse Repository System,” Proceedings of 34th Hawaii International Conference on System Sciences (HICSS-34), Jan. 3-6, 2001, pp 9065-9075.

[5] Desouza, Kevin, Awazu, Yukika, and Tiwana, Amri,. “Four Dynamics for Bringing Use Back into Software Reuse.” Communications of the ACM, 49(1), pp97-100, 2006.

[6] P. Wang and S.G. Shiva, "A Knowledge-Based Software Reuse Environment", IEEE Southeastern Symposium on System Theory, March 1994, pp276-280.

[7] Davor Čubranič, Gail C. Murphy, Janice Singer, and Kellogg S. Booth, “Hipikat: A Project Memory for Software Development”, IEEE Transactions on Software Engineering, vol. 31, no. 6, 2005, pp 446-465.

[8] Hans-Jorg Happel, Axel Korthaus, Stefan Seedorft, and Peter Tomczyk, “KOntoR: An Ontology-enabled Approach to Software Reuse” SEKE 2006, pp 349-354.

[9] M. Shaw, Architectural issues in software reuse: It's Not Just the Functionality, It's the Packaging, Proc IEEE Symposium on Software Reusability, April 1995, pp 3-6.

[10] W. Frakes, and Kyo Kang, “Software Reuse Research: Status and Future”, *IEEE Transactions on Software Engineering*, vol. 31, no. 7, 2005, pp 529-536.

[11] F. Buschmann et al., Pattern-Oriented Software Architecture. Chichester, UK; New York: Wiley, 1996.

[12] W. Frakes, R. Prieto-Diaz, and C. Fox, “DARE-COTS. A domain analysis support tool,” Computer Science Society, 1997. Proceedings, XVII International Conference of the Chilean 10-15, Nov. 1997, pp 73–77.

[13] J. Coplien, D. Hoffman, and D. Weiss, “Commonality and variability in software,” IEEE Software vol. 15, no 6, Nov.-Dec. 1998, pp 37-45.

[14] K. C. Kang, J. Lee, and P. Donohoe, “Featured Oriented Product Line Engineering,” IEEE *Software* vol. 15, No 6, pp 58-65, July-Aug. 2002.

[15] C. Atkinson, J. Bayer and D. Muthig, “Component-Based Product Line Development: The KobrA Approach,“ 1st International Software Product Line Conference, Denver, 2000. pp 289-310.

[16] H. Gomaa, Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, 2004.

[17] M. L. Griss, “Software Reuse at Hewlett-Packard,” Hewlett-Packard Company Technical Repost, 1991.

[18] M. L. Griss and A. Wosser, “Making Reuse Work at Hewlett-Packard.” IEEE Software, vol. 12, no. 1, Jan., 1995, pp. 105-107.

[19] A. Tomer, L. Goldin, T. Kuflik, E. Kimchi, and S. R. Schach, "Evaluating Software Reuse Alternatives: A Model and Its Application to an Industrial Case Study," IEEE Transactions on Software Engineering, vol. 30, no. 9, Sept., 2004, pp. 601-612.

[20] O. Lee, P. Banerjee, K. Lim, K. Kumar, J. Hillegersberg, and K. Wei, “Aligning IT components to Achieve Agility in Globally Distributed System Development,” Communications of the ACM, vol. 49, no 10, Oct. 2006, pp 49-54.

[21] Sebastian Schaffert, “IkeWiki: A Semantic Wiki for Collaborative Knowledge Management,” Proceedings of the 15th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2006, pp 388-396.

[22] B. Decker, E. Ras, J. Rech, B. Klein, and C. Hoecht, “Self-organized Reuse of Software Engineering Knowledge Supported by Semantic Wikis,” Workshop on Semantic Web Enabled Software Engineering (SWESE 2005), Galway, Ireland, 2005.

[23] I. Millard, A. Jaffri, H. Glaser, B. Rodriguez, “Using a Semantic MediaWiki with a Knowledge-Based Infrastructure,” The 15th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2006), Podebrady, Czech Republic, October 2006.

Sajjan G. Shiva is the Chair of the Computer Science Department at the University of Memphis in Tennessee. He was born in India where he received his B. S. degree in electrical engineering from Bangalore University in 1969. Then he received his M.S. (1971) and Ph.D. (1975) degrees in electrical engineering from Auburn University, Alabama.

Before working as a Professor at the University of Memphis, he was the President of Teksearch Inc. He was also a Technical Advisor in the Computer Technology Division of the United States Army Space and Strategic Defense Command and a consultant to industry and government. He is an emeritus professor of computer science at the University of Alabam in Huntsville. He is the author or coauthor of several books and scientific publications including Computer Organization, Design and Architecture (2007), Advanced Computer Architectures (2006), Introduction to Logic Design (1998), and Pipelined and Parallel Computer Architectures (1996). His research interests include software engineering, expert systems, data modeling, and distributed and parallel processing.

Dr. Shiva has been a fellow of the institute of Electrical and Electronics Engineers (IEEE) since 1993. He is the General Chair of the Second International Workshop on Advances and Innovations in Systems Testing organized by Systems Testing Excellence Program - STEP at The University of Memphis.

Lubna A. Shala is a Ph.D. student in the Computer Science Department at the University of Memphis in Tennessee. She received her B. S. and M.S degrees in computer and electronics engineering technology from The University of Memphis in 2003 and 2005 respectively. Her research interests include software engineering, natural language processing artificial intelligence, and software security.

Ms. Shala is a member of the Institute of Electrical and Electronics Engineers (IEEE) as well as the Association for Computing Machinery (ACM).

© 2008 ACADEMY PUBLISHER